# 01 | Mutex：如何解决资源并发访问问题？

2020-10-12 晁岳攀

Go 并发编程实战课 进入课程 >

**讲述：安晓辉**
时长 16:29 大小 15.10M

你好，我是鸟窝。

今天是我们 Go 并发编程实战课的第一讲，我们就直接从解决并发访问这个棘手问题入手。

说起并发访问问题，真是太常见了，比如多个 goroutine 并发更新同一个资源，像计数器；同时更新用户的账户信息；秒杀系统；往同一个 buffer 中并发写入数据等等。如果没有互斥控制，就会出现一些异常情况，比如计数器的计数不准确、用户的账户可能出现透支、秒杀系统出现超卖、buffer 中的数据混乱，等等，后果都很严重。

这些问题怎么解决呢？对，用互斥锁，那在 Go 语言里，就是 **Mutex。**

这节课，我会带你详细了解互斥锁的实现机制，以及 Go 标准库的互斥锁 Mutex 的基本使用方法。在后面的 3 节课里，我还会讲解 Mutex 的具体实现原理、易错场景和一些拓展用法。

好了，我们先来看看互斥锁的实现机制。

## 互斥锁的实现机制

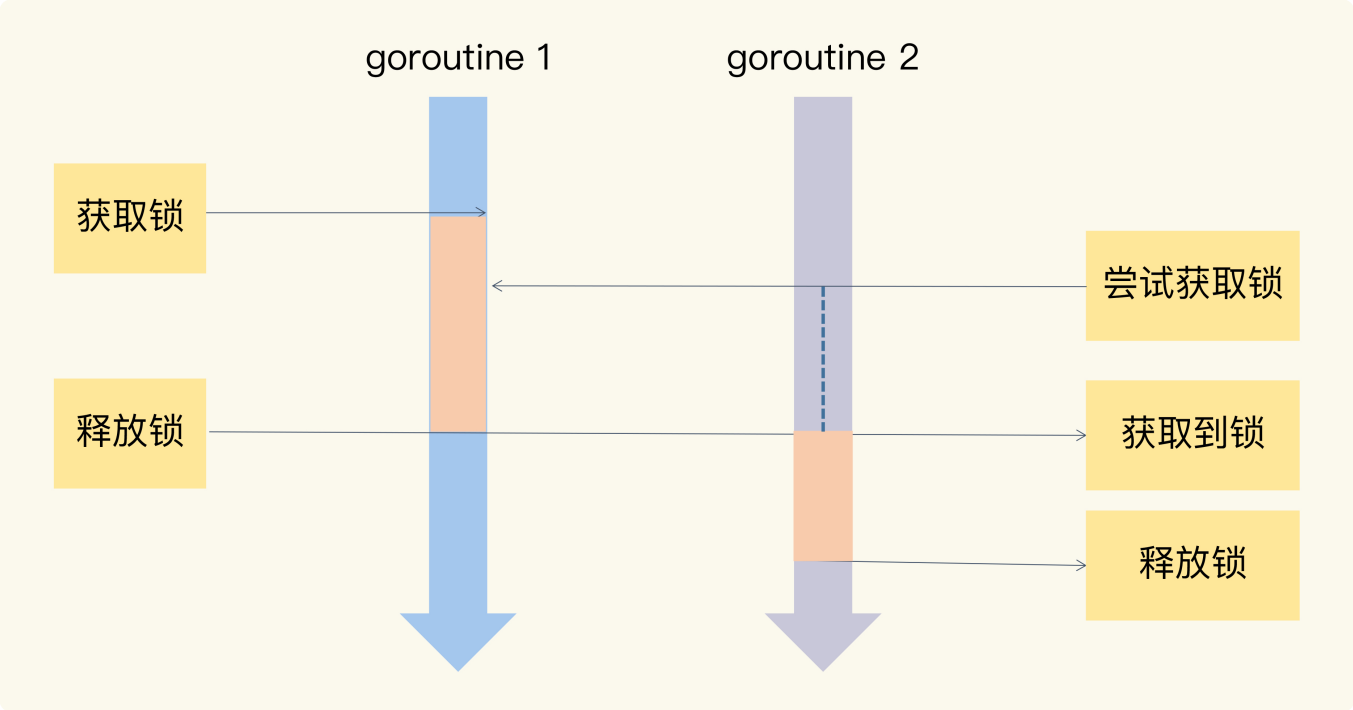
互斥锁是并发控制的一个基本手段，是为了避免竞争而建立的一种并发控制机制。在学习它的具体实现原理前，我们要先搞懂一个概念，就是**临界区**。

在并发编程中，如果程序中的一部分会被并发访问或修改，那么，为了避免并发访问导致的意想不到的结果，这部分程序需要被保护起来，这部分被保护起来的程序，就叫做临界区。

可以说，临界区就是一个被共享的资源，或者说是一个整体的一组共享资源，比如对数据库的访问、对某一个共享数据结构的操作、对一个 I/O 设备的使用、对一个连接池中的连接的调用，等等。

如果很多线程同步访问临界区，就会造成访问或操作错误，这当然不是我们希望看到的结果。所以，我们可以**使用互斥锁，限定临界区只能同时由一个线程持有**。

当临界区由一个线程持有的时候，其它线程如果想进入这个临界区，就会返回失败，或者是等待。直到持有的线程退出临界区，这些等待线程中的某一个才有机会接着持有这个临界区。




你看，互斥锁就很好地解决了资源竞争问题，有人也把互斥锁叫做排它锁。那在 Go 标准库中，它提供了 Mutex 来实现互斥锁这个功能。

根据 2019 年第一篇全面分析 Go 并发 Bug 的论文 Understanding Real-World Concurrency Bugs in Go，**Mutex 是使用最广泛的同步原语**（Synchronization primitives，有人也叫做**并发原语**。我们在这个课程中根据英文直译优先用同步原语，但是并发原语的指代范围更大，还可以包括任务编排的类型，所以后面我们讲 Channel 或者扩展类型时也会用并发原语）。关于同步原语，并没有一个严格的定义，你可以把它看作解决并发问题的一个基础的数据结构。

在这门课的前两个模块，我会和你讲互斥锁 Mutex、读写锁 RWMutex、并发编排 WaitGroup、条件变量 Cond、Channel 等同步原语。所以，在这里，我先和你说一下同步原语的适用场景。

> 共享资源。并发地读写共享资源，会出现数据竞争（data race）的问题，所以需要 Mutex、RWMutex 这样的并发原语来保护。
>
> 任务编排。需要 goroutine 按照一定的规律执行，而 goroutine 之间有相互等待或者依赖的顺序关系，我们常常使用 WaitGroup 或者 Channel 来实现。
>
> 消息传递。信息交流以及不同的 goroutine 之间的线程安全的数据交流，常常使用 Channel 来实现。

今天这一讲，咱们就从公认的使用最广泛的 Mutex 开始学习吧。是骡子是马咱得拉出来遛遛，看看我们到底可以怎么使用 Mutex。

## Mutex 的基本使用方法

在正式看 Mutex 用法之前呢，我想先给你交代一件事：Locker 接口。

在 Go 的标准库中，package sync 提供了锁相关的一系列同步原语，这个 package 还定义了一个 Locker 的接口，Mutex 就实现了这个接口。

Locker 的接口定义了锁同步原语的方法集：

复制代码

```
type Locker interface {
    Lock()
    Unlock()
}
```

可以看到，Go 定义的锁接口的方法集很简单，就是请求锁（Lock）和释放锁（Unlock）这两个方法，秉承了 Go 语言一贯的简洁风格。

但是，这个接口在实际项目应用得不多，因为我们一般会直接使用具体的同步原语，而不是通过接口。

我们这一讲介绍的 Mutex 以及后面会介绍的读写锁 RWMutex 都实现了 Locker 接口，所以首先我把这个接口介绍了，让你做到心中有数。

下面我们直接看 Mutex。

简单来说，**互斥锁 Mutex 就提供两个方法 Lock 和 Unlock：进入临界区之前调用 Lock 方法，退出临界区的时候调用 Unlock 方法**：

复制代码

```
func(m *Mutex)Lock()
func(m *Mutex)Unlock()
```

**当一个 goroutine 通过调用 Lock 方法获得了这个锁的拥有权后， 其它请求锁的 goroutine 就会阻塞在 Lock 方法的调用上，直到锁被释放并且自己获取到了这个锁的拥有权。**

看到这儿，你可能会问，为啥一定要加锁呢？别急，我带你来看一个并发访问场景中不使用锁的例子，看看实现起来会出现什么状况。

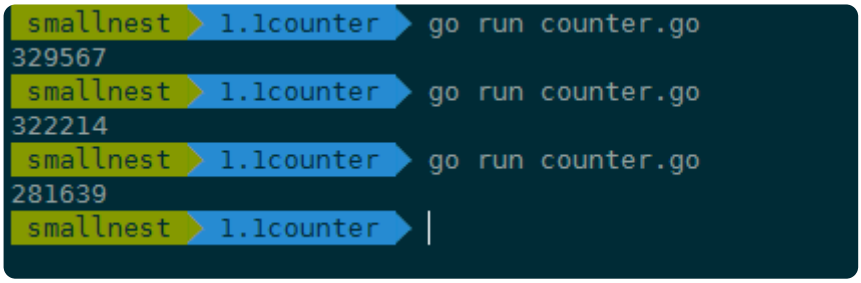
在这个例子中，我们创建了 10 个 goroutine，同时不断地对一个变量（count）进行加 1 操作，每个 goroutine 负责执行 10 万次的加 1 操作，我们期望的最后计数的结果是 10 * 100000 = 1000000 (一百万)。

复制代码

```
import (
        "fmt"
        "sync"
    )

    func main() {
        var count = 0
        // 使用WaitGroup等待10个goroutine完成
        var wg sync.WaitGroup
        wg.Add(10)
        for i := 0; i < 10; i++ {
            go func() {
                defer wg.Done()
                // 对变量count执行10次加1
                for j := 0; j < 100000; j++ {
                    count++
                }
            }()
        }
        // 等待10个goroutine完成
        wg.Wait()
        fmt.Println(count)
    }

```

在这段代码中，我们使用 sync.WaitGroup 来等待所有的 goroutine 执行完毕后，再输出最终的结果。sync.WaitGroup 这个同步原语我会在后面的课程中具体介绍，现在你只需要知道，我们使用它来控制等待一组 goroutine 全部做完任务。

但是，每次运行，你都可能得到不同的结果，基本上不会得到理想中的一百万的结果。




这是为什么呢?

其实，这是因为，**count++** 不是一个原子操作，它至少包含几个步骤，比如读取变量 count 的当前值，对这个值加 1，把结果再保存到 count 中。因为不是原子操作，就可能有并发的问题。

比如，10 个 goroutine 同时读取到 count 的值为 9527，接着各自按照自己的逻辑加 1，值变成了 9528，然后把这个结果再写回到 count 变量。但是，实际上，此时我们增加的总数应该是 10 才对，这里却只增加了 1，好多计数都被“吞”掉了。这是并发访问共享数据的常见错误。

复制代码

```
// count++操作的汇编代码
    MOVQ    "".count(SB), AX
    LEAQ    1(AX), CX
    MOVQ    CX, "".count(SB)
```

这个问题，有经验的开发人员还是比较容易发现的，但是，很多时候，并发问题隐藏得非常深，即使是有经验的人，也不太容易发现或者 Debug 出来。

针对这个问题，Go 提供了一个检测并发访问共享资源是否有问题的工具：race detector，它可以帮助我们自动发现程序有没有 data race 的问题。

**Go race detector** 是基于 Google 的 C/C++ sanitizers 技术实现的，编译器通过探测所有的内存访问，加入代码能监视对这些内存地址的访问（读还是写）。在代码运行的时候，race detector 就能监控到对共享变量的非同步访问，出现 race 的时候，就会打印出警告信息。

这个技术在 Google 内部帮了大忙，探测出了 Chromium 等代码的大量并发问题。Go 1.1 中就引入了这种技术，并且一下子就发现了标准库中的 42 个并发问题。现在，race detector 已经成了 Go 持续集成过程中的一部分。

我们来看看这个工具怎么用。

在编译（compile）、测试（test）或者运行（run）Go 代码的时候，加上 **race** 参数，就有可能发现并发问题。比如在上面的例子中，我们可以加上 race 参数运行，检测一下是不是有并发问题。如果你 go run -race counter.go，就会输出警告信息。

```
smallnest 1.1counter go run -race counter.go
==================
WARNING: DATA RACE
Read at 0x00c000126010 by goroutine 10:
  main.main.func1()
      /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.1counter/counter.go:16 +0x78

Previous write at 0x00c000126010 by goroutine 7:
  main.main.func1()
      /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.1counter/counter.go:16 +0x91

Goroutine 10 (running) created at:
  main.main()
      /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.1counter/counter.go:13 +0xe4

Goroutine 7 (running) created at:
  main.main()
      /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.1counter/counter.go:13 +0xe4
==================
==================
WARNING: DATA RACE
```

这个警告不但会告诉你有并发问题，而且还会告诉你哪个 goroutine 在哪一行对哪个变量有写操作，同时，哪个 goroutine 在哪一行对哪个变量有读操作，就是这些并发的读写访问，引起了 data race。

例子中的 goroutine 10 对内存地址 0x00c000126010 有读的操作（counter.go 文件第 16 行），同时，goroutine 7 对内存地址 0x00c000126010 有写的操作（counter.go 文件第 16 行）。而且还可能有多个 goroutine 在同时进行读写，所以，警告信息可能会很长。

虽然这个工具使用起来很方便，但是，因为它的实现方式，只能通过真正对实际地址进行读写访问的时候才能探测，所以它并不能在编译的时候发现 data race 的问题。而且，在运行的时候，只有在触发了 data race 之后，才能检测到，如果碰巧没有触发（比如一个 data race 问题只能在 2 月 14 号零点或者 11 月 11 号零点才出现），是检测不出来的。

而且，把开启了 race 的程序部署在线上，还是比较影响性能的。运行 go tool compile -race -S counter.go，可以查看计数器例子的代码，重点关注一下 count++ 前后的编译后的代码：

```
0x002a 00042 (counter.go:13)    CALL    runtime.racefuncenter(SB)
        ......
         0x0061 00097 (counter.go:14)    JMP     173
         0x0063 00099 (counter.go:15)    MOVQ    AX, "".j+8(SP)
         0x0068 00104 (counter.go:16)    PCDATA  $0, $1
         0x0068 00104 (counter.go:16)    MOVQ    "".&count+128(SP), AX
         0x0070 00112 (counter.go:16)    PCDATA  $0, $0
         0x0070 00112 (counter.go:16)    MOVQ    AX, (SP)
         0x0074 00116 (counter.go:16)    CALL    runtime.raceread(SB)
         0x0079 00121 (counter.go:16)    PCDATA  $0, $1
         0x0079 00121 (counter.go:16)    MOVQ    "".&count+128(SP), AX
         0x0081 00129 (counter.go:16)    MOVQ    (AX), CX
         0x0084 00132 (counter.go:16)    MOVQ    CX, "".autotmp_8+16(SP)
         0x0089 00137 (counter.go:16)    PCDATA  $0, $0
         0x0089 00137 (counter.go:16)    MOVQ    AX, (SP)
         0x008d 00141 (counter.go:16)    CALL    runtime.racewrite(SB)
         0x0092 00146 (counter.go:16)    MOVQ    "".autotmp_8+16(SP), AX
        ......
         0x00b6 00182 (counter.go:18)    CALL    runtime.deferreturn(SB)
         0x00bb 00187 (counter.go:18)    CALL    runtime.racefuncexit(SB)
         0x00c0 00192 (counter.go:18)    MOVQ    104(SP), BP
         0x00c5 00197 (counter.go:18)    ADDQ    $112, SP
```

在编译的代码中，增加了 runtime.racefuncenter、runtime.raceread、runtime.racewrite、runtime.racefuncexit 等检测 data race 的方法。通过这些插入的指令，Go race detector 工具就能够成功地检测出 data race 问题了。

总结一下，通过在编译的时候插入一些指令，在运行时通过这些插入的指令检测并发读写从而发现 data race 问题，就是这个工具的实现机制。

既然这个例子存在 data race 问题，我们就要想办法来解决它。这个时候，我们这节课的主角 Mutex 就要登场了，它可以轻松地消除掉 data race。

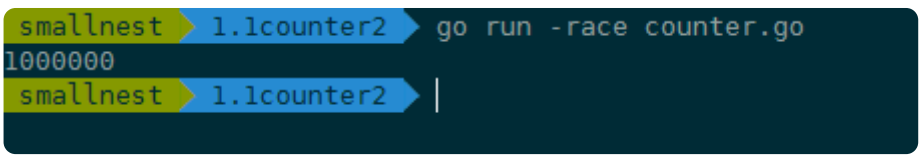
具体怎么做呢？下面，我就结合这个例子，来具体给你讲一讲 Mutex 的基本用法。

我们知道，这里的共享资源是 count 变量，临界区是 count++，只要在临界区前面获取锁，在离开临界区的时候释放锁，就能完美地解决 data race 的问题了。

复制代码

```
package main


    import (
        "fmt"
        "sync"
    )


    func main() {
        // 互斥锁保护计数器
        var mu sync.Mutex
        // 计数器的值
        var count = 0

        // 辅助变量，用来确认所有的goroutine都完成
        var wg sync.WaitGroup
        wg.Add(10)

        // 启动10个gourontine
        for i := 0; i < 10; i++ {
            go func() {
                defer wg.Done()
                // 累加10万次
                for j := 0; j < 100000; j++ {
                    mu.Lock()
                    count++
                    mu.Unlock()
                }
            }()
        }
        wg.Wait()
        fmt.Println(count)
    }
```

如果你再运行一下程序，就会发现，data race 警告没有了，系统干脆地输出了 1000000：



怎么样，使用 Mutex 是不是非常高效？效果很惊喜。

这里有一点需要注意：Mutex 的零值是还没有 goroutine 等待的未加锁的状态，所以你不需要额外的初始化，直接声明变量（如 var mu sync.Mutex）即可。

那 Mutex 还有哪些用法呢？

很多情况下，**Mutex 会嵌入到其它 struct 中使用**，比如下面的方式：

复制代码

```
type Counter struct {
    mu    sync.Mutex
    Count uint64
}
```

在初始化嵌入的 struct 时，也不必初始化这个 Mutex 字段，不会因为没有初始化出现空指针或者是无法获取到锁的情况。

有时候，我们还可以**采用嵌入字段的方式**。通过嵌入字段，你可以在这个 struct 上直接调用 Lock/Unlock 方法。

复制代码

```
func main() {
    var counter Counter
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            for j := 0; j < 100000; j++ {
                counter.Lock()
                counter.Count++
                counter.Unlock()
            }
        }()
    }
    wg.Wait()
    fmt.Println(counter.Count)
}


```

```
type Counter struct {
    sync.Mutex
    Count uint64
}
```

**如果嵌入的 struct 有多个字段，我们一般会把 Mutex 放在要控制的字段上面，然后使用空格把字段分隔开来。**即使你不这样做，代码也可以正常编译，只不过，用这种风格去写的话，逻辑会更清晰，也更易于维护。

甚至，你还可以**把获取锁、释放锁、计数加一的逻辑封装成一个方法**，对外不需要暴露锁等逻辑：

复制代码

```
func main() {
    // 封装好的计数器
    var counter Counter

    var wg sync.WaitGroup
    wg.Add(10)

    // 启动10个goroutine
    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            // 执行10万次累加
            for j := 0; j < 100000; j++ {
                counter.Incr() // 受到锁保护的方法
            }
        }()
    }
    wg.Wait()
    fmt.Println(counter.Count())
}

// 线程安全的计数器类型
type Counter struct {
    CounterType int
    Name        string

    mu    sync.Mutex
    count uint64
}

// 加1的方法，内部使用互斥锁保护
func (c *Counter) Incr() {
    c.mu.Lock()
    c.count++
```

```
35      c.mu.Unlock()
36  }
37
38  // 得到计数器的值，也需要锁保护
39  func (c *Counter) Count() uint64 {
40      c.mu.Lock()
41      defer c.mu.Unlock()
42      return c.count
43  }
```

## 总结

这节课，我介绍了并发问题的背景知识、标准库中 Mutex 的使用和最佳实践、通过 race detector 工具发现计数器程序的问题以及修复方法。相信你已经大致了解了 Mutex 这个同步原语。

在项目开发的初始阶段，我们可能并没有仔细地考虑资源的并发问题，因为在初始阶段，我们还不确定这个资源是否被共享。经过更加深入的设计，或者新功能的增加、代码的完善，这个时候，我们就需要考虑共享资源的并发问题了。当然，如果你能在初始阶段预见到资源会被共享并发访问就更好了。

意识到共享资源的并发访问的早晚不重要，重要的是，一旦你意识到这个问题，你就要及时通过互斥锁等手段去解决。

比如 Docker issue 37583、35517、32826、30696等、kubernetes issue 72361、71617等，都是后来发现的 data race 而采用互斥锁 Mutex 进行修复的。

## 思考题

你已经知道，如果 Mutex 已经被一个 goroutine 获取了锁，其它等待中的 goroutine 们只能一直等待。那么，等这个锁释放后，等待中的 goroutine 中哪一个会优先获取 Mutex 呢？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

31 人觉得很赞 | 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 开篇词 | 想吃透Go并发编程，你得这样学！

下一篇 02 | Mutex：庖丁解牛看实现

## 精选留言 (32)

写留言

**Alexdown**
2020-10-12

等待的goroutine们是以FIFO排队的
1）当Mutex处于正常模式时，若此时没有新goroutine与队头goroutine竞争，则队头goroutine获得。若有新goroutine竞争大概率新goroutine获得。
2）当队头goroutine竞争锁失败1ms后，它会将Mutex调整为饥饿模式。进入饥饿模式后，锁的所有权会直接从解锁goroutine移交给队头goroutine，此时新来的goroutine...

展开

1 48

**A**
2020-10-12

看了下作者的发量，果断入手

展开

41

**我来也**
2020-10-12

课后思考题：
分享两篇文章吧（我还没看完😂）

鸟叔的：sync.mutex 源代码分析
https://colobu.com/2018/12/18/dive-into-sync-mutex/...

展开

3 11

**Remember九离**
2020-10-12

这个课程我想一直跟着走，不单单只是看，我想在吸收课程精华的同时，也进行独立的思考，然后输出，总结，加深理解。因此创建了一个仓库，欢迎大家在学习的同时一起做到手中有码，项目地址:https://github.com/wuqinqiang/Go_Concurrency

展开

1 6

**Daiver**
2020-10-12

go的goroutine 调度中，内部维护了队列，goroutine在抢占锁的时候，会自旋一段时间，如果抢占失败，这个goroutine会被放到一个FIFO队列中，一般来说，锁释放时，会优先唤醒队头的goroutine，即队头的goroutine优先获得Mutex。

展开

5

**pedro**
2020-10-13

大家都已经解答了，就不重复了。这里给一些不熟悉 go 需要的同学补充一下，go 语言查看汇编代码命令:
go tool compile -S file.go
对于文中 counter 的例子可以过度优化一下，那就是获取计数的 Count 函数其实可以通过读写锁，也就是 RWMutex 来优化一下。

展开

作者回复: 你居然“剧透”第五讲的内容

1 4

**chapin**
2020-10-13

个人希望 窝大佬可以多分析一些源码。

展开

作者回复: 明天一讲剖析mutex

3

**Panda**
2020-10-20

推荐一个工具 Chronos - A static race detector for the go language
https://github.com/amit-davidson/Chronos

展开

2

**ZY**
2020-10-17

有两种情况
1. 如果当前有协程进入自旋模式，当前协程会成功获取到锁
2. 如果没有协程进入自选模式，释放锁的协程会释放的信号量会成功唤醒等待队列中的协程，该卸程会成功获取到锁，并且把等待计数器减1.
...

展开

作者回复: 进入自旋不一定会获取到锁。
饥饿模式不一定改变，看文章。只有等待时间小于阈值或者无等待者时才会改变模式

2

**Panmax**
2020-10-18

原文中关于 race detector 的介绍有两句话前后矛盾，老师可否解释一下：

前边说：在编译（compile）、测试（test）或者运行（run）Go 代码的时候，加上 race 参数，就有可能发现并发问题。

…

展开

作者回复: 那借这个问题解答一下吧。编译的时候不能发现data race,但是编译的时候可以开启race参数，这样编译后的程序在运行时就可以data race问题了。你看到很仔细，希望这个解答能把这段解释清楚。
另外，绝对不要把带race参数编译的程序部署到线上。

1

**新味道**
2020-10-13

func (c *Counter) Incr() { c.mu.Lock() c.count++ c.mu.Unlock()}// 得到计数器的值，也需要锁保护func (c *Counter) Count() uint64 { c.mu.Lock() defer c.mu.Unlock() return c.count}

问题：Incr()里为什么不用defer c.mu.Unlock() ？

展开

作者回复: ����������看的仔细。也可以。defer的耗时已经非常小了，我专栏中有讲到。

1 1

不瘦二十斤不改头像

**jeffery**
2020-10-27

对于go小白来说怎么快速的进阶！看鸟叔.发型.必定课程干货满满

**Kcaco**
2020-10-26

思路清晰，学到很多！！！

展开

**无名氏**
2020-10-22

每篇文章多品几遍

展开

**Bug Killer**
2020-10-21

Counter 中的 CounterType 和Name 属性为啥首字母要大写，是有啥特殊意义吗

1

**悦悦**
2020-10-21

在运行的时候，只有在触发了 data race 之后，才能检测到，如果碰巧没有触发（比如一个 data race 问题只能在 2 月 14 号零点或者 11 月 11 号零点才出现），是检测不出来的。我运行时，开启data race 不太理解，2月14号是检查不出来！

展开

作者回复: 相关的代码只有被触发，并且碰巧有data race才能被检测出来

**houseme**
2020-10-20

打卡

展开

**不求闻达**
2020-10-19

grouting 是以FIFO队列顺序来进入临界区。这只是正常模式。如果是饥饿模式就不一样了。

**Panda**
2020-10-19

同时更新用户的账户信息 这种场景 实际生产环境用分布式锁是不是更合适 例如 Redis 分布式锁 ZK 分布式锁

展开

**oCupJS**

2020-10-18

打卡

展开